An Offline Algorithm for Dimension-Bound Analysis

Paul A.S. Ward
Shoshin Distributed Systems Group
Department of Computer Science
University of Waterloo
pasward@styx.uwaterloo.ca

Abstract

The vector-clock size necessary to characterize causality in a distributed computation is bounded by the dimension of the partial order induced by that computation. In an arbitrary distributed computation the dimension can be as large as the width, which in turn can be as large as the number of processes in the computation. Most vector clock algorithms, and all online ones, simply use a vector of size equal to the number of processes. In practice the dimension may be much smaller. It is the purpose of this paper to provide empirical evidence that the dimension of various distributed computations is often substantially smaller than the number of processes. We have found that typical distributed computations, with as many as 300 processes, have dimension less than 10. To achieve this quantification we developed various theorems and algorithms which we also describe.

1 Motivation

An important problem in distributed systems is monitoring and debugging distributed computations. This problem is hard because events in the computation can be concurrent. The events form a partial order, not a total one. While displaying this partial order can be a useful debugging aid, for any non-trivial computation it is not possible to display the whole partial order. As a result, distributed debugging systems such as POET [13] need to provide much more than just a drawing. It is necessary to intelligently scroll around the display [20], search for interesting patterns [10], compute differences between subsequent executions of a computation [8], detect race conditions [23], determine appropriate abstractions to provide higher level views [12], and so on. To perform these operations it is frequently necessary to determine event precedence. That is, given two events, it is necessary to be able to efficiently determine if they are ordered or if they are concurrent.

Event precedence may be determined in several ways, depending on how the partial order is represented. If the partial order is stored as a directed acyclic graph, then precedence determination is a constant time operation. This is because the partial order is transitively closed and so there is an edge between any two events that are ordered. However, the space consumption for this method is unacceptably high. If, on the other hand, the transitive reduction of the partial order is used, much less space is needed. Unfortunately, this requires a (potentially quite slow) search operation on the graph to determine precedence. To compensate for this deficiency a vector clock [3, 15] is associated with each event. If the processes in which the events occur are known, then it is possible to determine precedence with vector clocks in constant time.

The size of a vector clock necessary to capture causality is bounded by the dimension of the partial order induced by the computation. Charron-Bost [2] has shown that the dimension can be as large as the width, and all online vector clocks developed to date require a vector with size equal to the number of processes (which forms an upper bound on the width). Since we need to associate such a vector clock with every event in the computation, we are substantially constrained in the number of processes that we can observe. In POET we have found that due to this limitation we can handle at most a few-hundred processes.

The Charron-Bost proof, while true, relies on a very specific distributed computation. It was our belief, and this paper provides empirical evidence to support it, that this computation, or variations on it, simply does not occur in practice, and that those computations that do occur in practice, tend to have a much lower dimension than the number of processes involved. In this paper we provide an algorithm for estimating the dimension of the partial order induced by a distributed computation. While we do not yet have a bound on the quality of the algorithm, the results we present show that the dimension of typical computations is substantially smaller than the number of processes involved in the computation. It is therefore possible to use a much

smaller vector clock than is used at present for any offline analysis. We are still working on developing an online vector clock whose size is bounded by dimension, not by the number of processes.

In the remainder of this paper we will specify first the formal model of distributed computation and why this leads to a problem with vector-clock size. In Section 3 we will describe the theorems and algorithms we developed to determine the dimension bound of distributed computations. We then discuss the results we achieved from this after executing the algorithms over several computations in various parallel, concurrent and distributed environments. Finally we indicate what work remains to be completed to achieve online vector clocks whose size is dimension-bounded

2 Background

We use the standard model of distributed systems, initially defined by Lamport [14]: a distributed system is a system comprising multiple sequential processes communicating via message passing. Each sequential process consists of three types of events, send, receive and unary, totally ordered within the process. A *distributed computation* is the partial order formed by the "happened before" relation over the union of all of the events across all of the processes. We will refer to the set of all events with E, the set of events within a given process by E_i (where i uniquely identifies the process) and an individual event by e_i^j (where i identifies the process and j identifies the event's position within the process). Then the Lamport "happened before" relation ($\rightarrow \subseteq E \times E$) is defined as the smallest transitive relation satisfying

1. $e_i^j \rightarrow e_i^l$ if $j < l$
2. $e_i^j \rightarrow e_k^l$ if e_i^j is a send event and e_k^l is the corresponding receive event

Events are concurrent if they are not in the "happened before" relation.

$$e_i^j \parallel e_k^l \iff e_i^j \not\rightarrow e_k^l \wedge e_k^l \not\rightarrow e_i^j \qquad (1)$$

Each event in the computation has an associated vector clock for event precedence determination. Since we are working in a debugging context, the vector clock is not a part of the computation. Rather, it is computed separately by the debugging agent. We must now describe some partial-order terminology in order to explain why these vectors are of size equal to the number of processes in the computation.

2.1 Partial-order terminology

The following terminology is due to Trotter [21]. A *strict partial-order* (or partially-ordered set, or poset) is a pair (X, P) where X is a finite set[1] and P is an irreflexive,[2] antisymmetric and transitive binary relation on X. A *subposet*, $(Y, P|_Y)$, is a poset whose set Y is a subset of X, and whose relation $P|_Y$ is the restriction of P to the subset. An *antichain* is any completely unordered poset. The *width* of a poset is the longest antichain contained in that poset. In the context of a distributed computation, the width must be less than or equal to the number of processes.

An *extension*, (X, Q), of a partial order (X, P) is any partial order that satisfies

$$(x, y) \in P \Rightarrow (x, y) \in Q$$

If Q is a total order, then the extension is called a *linear extension*. If (Y, R) is an extension of the subposet $(Y, P|_Y)$ of (X, P), then (Y, R) is said to be a *subextension* of (X, P). A *realizer* of a partial order is any set of linear extensions whose intersection forms the partial order. The *dimension* of a partial order is the cardinality of the smallest possible realizer.

Finally, we say that "y is an *immediate predecessor* of z" or "z is an *immediate successor* of y" if y precedes z and there is no intermediate element in the partial order between y and z.

$$y <: z \iff y \rightarrow z \wedge (\nexists_w y \rightarrow w \wedge w \rightarrow z) \qquad (2)$$

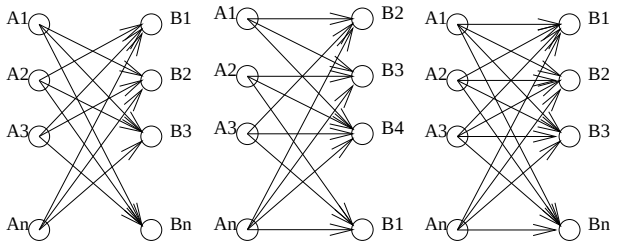
2.2 The vector-clock size problem

There are two reasons why vector clocks have size equal to the number of processes in the distributed computation. The first reason is algorithm availability. The best online algorithms for vector clocks are typically Fidge/Mattern variants [3, 15], which require a vector of size equal to the number of processes. While there is an alternate, the Ore timestamp (to be discussed in Section 2.3), whose size is bounded by the dimension of the partial order, it is an offline technique.

The second reason is more theoretical. To capture precedence in a partial order it is necessary to have a vector (or equivalent) of size equal to the dimension of that partial order [16]. Further, the dimension of a partial order can be as large as the width [21]. Crown $\mathbf{S}_n^0$ is the standard example of such a partial order, and is shown in Figure 1(a). By shifting each B_i element of this partial order we create the distributed computation shown in Figure 1(b). While this computation does not violate our model of distributed

[1] Since we are modeling distributed computations, all of the sets will be finite.

[2] If P is reflexive, it is a *partial order* rather than a *strict partial order*. The difference is not an important point in our context. The "happened before" relation, as defined by Lamport [14], is irreflexive, and so we use strict partial orders.



(a) Crown $\mathbf{S}_n^0$ (b) Shifted crown (c) Broadcast

Figure 1. Crown and broadcast partial orders

computation, it is unusual in that it requires both multicast send operations and corresponding multi-receive operations. While both of these operations do have real systems counterparts, it is important to emphasize that neither is necessary. Charron-Bost's study [2] relies only on point-to-point computation, though the sample computation produced is, in essence, the same as that of the Figure 1(b). Both correspond to all processes sending a message to all other processes, with the exception of their left neighbour.

The limitation of the Charron-Bost proof is precisely in the nature of what the crown $\mathbf{S}_n^0$ distributed computation represents. In practical terms, this is not a realistic distributed computation. In addition, the more likely computation, in which each process broadcasts a message to all other processes (shown in Figure 1(c)), has dimension 2. It is therefore the objective of this paper to determine a bound on the dimension of actual distributed computations, rather than theoretical ones that are doubtful to ever occur in practice. This bound would represent a more accurate requirement for the size of timestamps necessary to capture causality.

Before we describe how we compute the dimension bound, and what our results are for typical distributed computations, we wish to justify that there are in practice alternate timestamps that are more compact than Fidge/Mattern vector clocks. We will also discuss work related to our own.

2.3 Ore timestamps

There is an alternate timestamp to the Fidge/Mattern vector clock, whose size is bounded by the dimension, not the width, of the partial order. This is the Ore timestamp [16]. Given a realizer for partial order (X, P) the timestamp associated with each $x \in X$ is simply the vector of its position in the various linear extensions that form the realizer. It is then straightforward to see that causality between two events, x and y can be determined by comparing the various elements in the vectors of the two events. If they are all less, then precedence is established. If some are less and some are greater then concurrency is established. While this timestamp is offline, and it is beyond the scope of this paper to correct this deficiency, it does indicate that timestamps whose size is bounded by the dimension, not the width, of the partial order are not only possible in theory but exist in practice.

2.4 Related work

Before preceding to describe how we computed dimension bounds, we will briefly describe some of the work related to our own. There are essentially two categories of related work. The first group are those who are developing systems for visualizing parallel and distributed systems in a process-time fashion. Such work includes GOLD [17], ParaGraph [9] and our own system, POET [13]. Other than our own, these systems tend to use vector-clocks or variants within the computation, rather than have the information sent to a central server which computes the vector-timestamps for visualization purposes. These systems tend to take the approach of just using what is presently available, and not being concerned with substantial scalability. The GOLD system uses dependency vectors, developed by Fowler and Zwaenepoel [5], which will be size $O(n)$ by the time they are attached to individual events. ParaGraph has the ability to provide space-time diagrams, but there is no attempt to determine causality. It is merely a visualization, based on, possibly badly synchronized, local clocks It is up to the user to trace the dependencies. Indeed, the authors acknowledge that the size would not scale well beyond 128 processes, as the display becomes too cluttered. POET uses standard Fidge/Mattern timestamps, and so it too requires vectors of size equal to the number of processes.

The second group are those who are trying to reduce the vector-clock size, but in the context of maintaining vector-clocks by the processes involved in the computation. There are three main algorithms in this area. The first technique, due to Singhal and Kshemkalyani [18], sends a differential vector of what has changed since the last communication to the receiving process. It requires two additional vectors of size n at each process to recover the timestamp information. The second method, due to Fowler and Zwaenepoel [5], is to maintain only direct dependency vectors. That is, only the scalar time at the local process is transmitted to a receiver. A graph search is needed to go from this information to full precedence information. The third algorithm, due to Jard and Jourdan [11], creates a pseudo-direct relation that is stronger than direct dependency. The basic problem with all of these techniques is that they do not apply in the debugging or monitoring context. In the context of maintaining vector time in a distributed computation, they all succeed, to one degree or another, in reducing the message-size overhead required. In our context, we must maintain

vectors for all events at all times. What is relevant is not, therefore, the message-size overhead, as we send no messages but rather computer the vectors separately from the computation. What is relevant is the amount of information necessary to determine precedence.

Finally, there are no performance results that we are aware of concerning the actual behaviour of the various systems of the first group or algorithms of the second group in the context of large numbers of processes. In this respect, our paper is unique.

3 Bounding the dimension

In this section we will describe the algorithms we have developed to compute the dimension of the partial order. This is done in a two-phase process. We first compute the critical pairs of the partial order, and then we create a set of extensions that reverses those pairs. We will first describe the formal justification for this approach, and then describe the two phases. Finally we will provide some analysis of the algorithm.

Computing the exact dimension of a partial order is known to be NP-hard for any partial order of dimension greater than two [22]. We therefore approached the problem by attempting to simply bound the dimension. For our purposes, an order of magnitude difference between the dimension bound and the number of processes would be sufficient to justify proceeding. It was then necessary to develop an algorithm to achieve a reasonable bound in a reasonable amount of time. Rather than take the direct approach of generating linear extensions and then determining if they formed a realizer we chose an indirect route based on the concept of *critical pairs*.

Definition 1 (critical pair) (x, y) *is a critical pair of partial order* (X, P) *if* $x \parallel y$ *and* $(X, P \cup \{(x, y)\})$ *is a partial order.*

An equivalent definition is $(x, y) \in \mathrm{CP}$ if and only if

$$x \parallel y \wedge \forall_{z \in X} z \rightarrow x \Rightarrow z \rightarrow y \wedge y \rightarrow z \Rightarrow x \rightarrow z$$

where CP is the set of all critical pairs of the partial order (X, P). The significance of critical pairs, as regards dimension, is in the following theorem [21].

Theorem 1 *The dimension of a partial order is equal to the least number of subextensions of that partial order required to reverse all of its critical pairs.*

A critical pair (x, y) is said to be reversed by a set of subextensions if one of the subextensions in the set contains $y \rightarrow x$. In simpler terms, it is sufficient to simply reverse the critical pair events. Specifically, all events that are not part of a critical pair may be ignored. Further, not every subextension need contain all critical pairs. Note also, that it is not necessary for the subextensions to be linear. They merely have to reverse the critical pairs.

The approach we have then taken to the problem of bounding the dimension is to first compute all of the critical pairs of the partial order (a polynomial-time problem) and then create extensions that reverse these critical pairs (an NP-hard problem).

3.1 Computing critical pairs

While it is possible to use the given definition of critical pairs to compute the set of all critical pairs, any such algorithm would likely be very inefficient. To achieve reasonable performance in the computation it is necessary to develop an association of critical pairs with the relations that hold for the partial order. To this end, we define the sets $\mathrm{leastConcurrent}(e)$ and $\mathrm{greatestConcurrent}(e)$.

Definition 2 (leastConcurrent(e)) *The set of events that are* leastConcurrent *to an event* e *are those events that are concurrent with* e *and which have no predecessor which is also concurrent with* e.

In formal terms, $\mathrm{leastConcurrent}(e)$ is the set:

$$\left\{ e_l \mid e_l || e \wedge \left(\nexists_{e'_l} \, e'_l || e \wedge e'_l \rightarrow e_l \right) \right\} \tag{3}$$

Likewise the set $\mathrm{greatestConcurrent}(e)$ is:

$$\left\{ e_g \mid e_g || e \wedge \left(\nexists_{e'_g} \, e'_g || e \wedge e_g \rightarrow e'_g \right) \right\} \tag{4}$$

This then leads to the following theorem.

Theorem 2 (x, y) *form a critical pair if and only if*

$$x \in \mathrm{leastConcurrent}(y) \wedge y \in \mathrm{greatestConcurrent}(x)$$

This theorem enabled the development of the following algorithm.

```
1:∀_y y ∈ Events in Computation {
2:     l_C ← leastConcurrent(y)
3:     ∀_x x ∈ l_C {
4:         g_C ← greatestConcurrent(x)
5:         if (y ∈ g_C) {
6:             (x, y) is a critical pair
7:         }
8:     }
9: }
```

Some comments should be made about this algorithm. First, it is not obvious from the above why we perform the computation in what appears to be the reverse order. That is, we take each event as a possible second element

of a critical pair, rather than a first element. The reason has to do with the relative cheapness with which we can compute the $\mathrm{leastConcurrent}$ set versus the comparative expense of computing the $\mathrm{greatestConcurrent}$ set. We implemented our algorithm as a client to the POET server. As such we have access to Fidge/Mattern timestamps for each event. An important property of these timestamps is that they give the set of events that are the greatest predecessors in each process to the event for which they form the timestamp. What this means is that we can efficiently compute the $\mathrm{leastConcurrent}$ set of an event as follows.

```
1: leastConcurrent(e) {
2:     l_C ← timestamp(e) + 1
3:     ∀_x x ∈ l_C {
4:         if (x ∦ e) {
5:             l_C ← l_C − x
6:         }
7:     }
8:     ∀_x x ∈ l_C {
9:         ∀_y y ∈ l_C {
10:            if (x → y) {
11:                l_C ← l_C − y
12:            }
12:        }
13:    }
14:    return l_C
15: }
```

In words, to compute $\mathrm{leastConcurrent}(e)$ we start with the timestamp of e, that is, those events that are the greatest predecessors to e in their respective processes. We advance this timestamp by one; that is, we increment each element of the timestamp. This now represents a set of events that are either concurrent or successors to e. We refer to this set as the set of potentially least concurrent events of e. We remove from this set any event that is not concurrent with the event e and any event that is preceded by some other event within the set. This leaves those events that are $\mathrm{leastConcurrent}(e)$. This computation costs $O(w^2)$ where w is the width of the partial order (that is, the number of processes in the computation). The reason is that the number of iterations of the outer and inner loops of the nested loop is equal to the cardinality of the lC set, which can be as large as the width.

The $\mathrm{greatestConcurrent}(e)$ set is more expensive to calculate. To compute it we start with the greatest predecessors (*i.e.* the Fidge/Mattern timestamp) of e. We iterate along each process' set of events until we reach the event immediately prior to a successor event to e, or we come to the last event in that process. This yields the potentially greatest concurrent set. We then remove all events in this set that are not concurrent with e or that precede any other events in the set. This leaves the $\mathrm{greatestConcurrent}(e)$ set. The cost of this operation can be quite substantial, since we must compare with an arbitrary number of successors to the greatest predecessors of e. This motivated the discovery of the following theorem.

Theorem 3 *If two events, x and y, are concurrent then either $y \in \mathrm{greatestConcurrent}(x)$ or x precedes some immediate successor of y.*

Since we will not compute $\mathrm{greatestConcurrent}(x)$ until we know that $x \in \mathrm{leastConcurrent}(y)$ we know that $x \parallel y$. What this means in practice is that we do not compute the $\mathrm{greatestConcurrent}(x)$ set at all, but rather simply check whether x is a predecessor of all immediate successors of y. If it is, then (x, y) form a critical pair. Thus the algorithms becomes:

```
1: ∀_y y ∈ Events in Computation {
2:     l_C ← leastConcurrent(y)
3:     ∀_x x ∈ l_C {
4:         l_S ← {s | y <: s}
5:         if (∀_s s ∈ l_S ⇒ x → s) {
6:             (x, y) is a critical pair
7:         }
8:     }
9: }
```

Since we only support point-to-point communication, the set of events that are immediate successors to an event has cardinality less than 3. This means that the cost to compute the critical pairs is $O(w^2)$ per event or $O(w^2 n)$, where n is the total number of events, for the whole distributed computation.

3.2 Reversing critical pairs

Building the minimum number of extensions that reverses the critical pairs of a partial order is NP-hard for dimension greater than two [22]. Instead we propose a reasonably efficient algorithm that will not give an optimal solution, but will provide an upper bound on the minimum number of extensions necessary (that is, the dimension). Insofar as the dimension-bound we compute is small, it is a satisfactory tradeoff. To achieve this we developed a two-step algorithm. First we select the desired extension in which we will reverse the current critical pair and then we insert it into that extension.

In accordance with Theorem 1, it is sufficient to develop subextensions that reverse the critical pairs of the partial order. We do not have to insert all critical pairs in all subextensions. We merely have to find one subextension that will reverse the critical pair. Each subextension then is composed of some reversed critical pair events and nothing else. Note that this approach would not be sufficient for generating Ore

timestamps. It is insufficient as each subextension contains just a subset of the events of the partial order, not all of the events. Further, Ore timestamps require that the extensions are linear. Note also that using subextensions, rather than extensions, has implications on how we can insert critical pairs into the subextension.

To understand the algorithm we must define what it means to insert a critical pair into an subextension. It means that we can add the events of the critical pair, such that they are reversed, and that it violates neither the partial order, nor the additional constraints that the reversal requires. It may also be the case, if the insertion algorithm is not optimal, that a subextension rejects the critical pair even though it did not violate these conditions, but rather violated some aspect of the structure in which the extension was kept.

It is, perhaps, helpful to consider a simple example. Suppose we have a partial order consisting solely of two concurrent events, a and b. It has critical pairs (a, b) and (b, a). Once (a, b) has been inserted into a subextension, that subextension must reflect the constraint $b \rightarrow a$. As such (b, a) cannot be inserted into that subextension, since it would require the subextension to reflect $a \rightarrow b$.

We say that a subextension *accepts* a critical pair if the critical pair may be inserted into that subextension. A subextension *rejects* a critical pair if it does not accept it. Since a subextension may reject a critical pair, insertion into a subextension may fail. Therefore the first step of the algorithm must have a strategy for selecting an alternate subextension in which to place the critical pair. We can now describe the specific algorithms used for the two steps.

The algorithm we used for the extension-selection step is a simple greedy one. We insert the current critical-pair events into the first extension that will accept it. In the event that all current extensions reject the critical pair, we create a new extension, containing no events, that must, by definition, accept the critical pair. The critical pair is inserted into the new extension. Thus, the first step algorithm is:

```
 1: insert(x,y) {
 2:     for (i = 0; i ¡ numberExtensions; ++i) {
 3:         if (insert(extension[i], x, y)) {
 4:             return
 5:         }
 6:     }
 7:     create(extension[numberExtensions])
 8:     insert(extension[numberExtensions], x, y)
 9:     ++numberExtensions
10:     return;
11: }
```

3.2.1 Subextension insertion

For the second step of the algorithm, we had to define a method for inserting critical pairs into an subextension. The initial algorithm that we developed was a greedy one that worked on the principle "place the event before the first event it *must* precede." While this approach produces some promising results, it also produces some spectacularly bad ones.

We therefore decided to develop an optimal solution to this second step. We maintain a directed acyclic graph for each subextension. To add a critical pair we add the two events in turn and then determine if the graph is still acyclic. If it is acyclic we have accepted and inserted the critical pair. If it is not, we reject the critical pair, and remove the evidence of the addition. This method proved to be acceptable, as we can see in Section 4.

The data structure for a given node of the DAG representing event e maintains the following information: the vector timestamp of e, the sets $\{\lambda \mid (\lambda, e) \in \mathrm{CP}_S\}$ and $\{\rho \mid (e, \rho) \in \mathrm{CP}_S\}$ (where $\mathrm{CP}_S \subseteq \mathrm{CP}$ is the subset of the critical pairs that this subextension S has reversed), and a set of pointers to some of the successors to e in the DAG. The actual successors pointed to will depend on the order in which events are inserted. It is not typically the minimum set of successors needed, but neither is it the full transitive closure.

We now define event precedence between two DAG events as follows.

$$x \prec_S y \iff x \rightarrow y \vee (y, x) \in \mathrm{CP}_S$$

where CP_S is the set of critical pairs that are reversed by subextension S, as defined above. Event x precedes y in subextension S if and only if x precedes y in the partial order or (y, x) is a critical pair that is reversed by this subextension. Note that if events a and b are in a subextension S and (a, b) form a critical pair this does not necessarily imply that $(a, b) \in \mathrm{CP}_S$. The pair (a, b) is only in CP_S if the subextension S has accepted, that is reversed, it. A simple example of this case is if the subextension S already contains the critical pair (b, a).

A second significant aspect of this definition is that it does not capture transitivity. Thus if z is an immediate successor to x and both are concurrent to y, with (x, y) forming a critical pair, then $y \prec x$ and $x \prec z$ but $y \not\prec z$. Appropriate transitivity can only be captured by traversing the DAG.

We designate a special node *root* with the property that $\forall_x$ *root*$\prec_S x$. The root node enables us to enter the DAG at a single point, rather than, potentially, multiple concurrent points.

The insertion algorithm is then as follows. We traverse the DAG in a depth-first search order, starting at the root, comparing the event being inserted with the current node.

We determine if the event equals, succeeds, precedes or is concurrent with the current node. Only one of these must be the case, and if more than one has occurred, we abort the insertion. It has failed. If the event precedes or equals the current node, then it must not succeed or equal any successors to the current node. We therefore set a mustNotSucceedOrEqual variable to this effect. If this variable has been set before, and the event succeeds or equals the current node we will abort the insertion.

If the event succeeds the current node, then we may have to add successor pointers to the event from the current node. We have to add a successor pointer from the current node to the event if it precedes any of the successors to the current node, or if it is concurrent with all of the successors. In the former case we will also add a pointer from the event to the successor node that it precedes, and remove the pointer from the current node to the successor node, since there is a path through the event..

We repeat these operations for the successor nodes of the current node, in depth-first order. The mustNotSucceedOrEqual variable is stored and recovered from the depth-first search stack.

To traverse the DAG there is a "mark" integer associated with each node. Before each traversal we increment the mark value in the root. As we visit a node we set the value of mark at that node to value of the mark at the root. We only visit a node if the mark is less than the root mark value. Since we simply use an integer for this mark, if we ever reach about half-a-billion critical pairs this value will wrap. This defect can be easily fixed when the need arises.

To enable us to undo any changes we make, before we make any change we create a copy of the DAG node. We only do this if we do not yet have a copy. After successful insertion of both events of the critical pair we traverse the DAG completely to commit the changes. That is, we traverse the DAG and delete the copy at any node that has one. If the insertion of the events is unsuccessful, we traverse the DAG and abort the changes. That is, we traverse the DAG and, wherever there was a copy made we restore that copy over the existing node.

3.3 Algorithm analysis

We now turn to studying the quality of the algorithm. There are two aspects to this. We wish to study the efficiency of the algorithm and we would like to know how tight dimension-bounds are that it produces, given that the algorithm does not produce the optimal bound.

We have already seen that the computation cost to determine the critical pairs is $O(w^2)$ per event. The cost of reversing a critical pair is in the worst case equal to the cost of attempting to insert it into all of the extensions. We presume that each extension in turn rejects the insertion. If the dimension bound produced is d then there will be d such attempts. The cost of each insertion attempt is, again in the worst case, $O(bn)$ where n is the total number of events and b is the branching factor of the extension DAG, since each successor must be explicitly compared with the event being inserted and all nodes in the DAG must be examined. Thus, the total cost of the algorithm is $O(bdw^2n^2)$. This bound is probably not tight, and we already know of ways in which we could reduce the algorithm complexity, though this was not our prime concern.

The quality of the dimension-bound produced is something that we are currently investigating. We know that the bound we produce will never exceed w as to do so would require more than w mutually conflicting critical pairs. This would imply that the dimension could exceed the width, which we have already seen is not possible. We also know that the algorithm will never give a figure that is less than the dimension of the computation. In this respect it is conservative. Given this fact, and since the dimension bound produced tends to be in single digits, the algorithm is in general producing an answer that is within a factor of 5 of the actual dimension (it is trivial to determine that all of the computations have a dimension of at least 2).

The only step in the algorithm that is not optimal (from a dimension-bound perspective) is the subextension selection step. We have therefore experimented with varying the ordering in which subextensions are selected. There are several variations we have considered, which largely fall into two camps: random selection and deterministic selection. The random selection techniques included a random selection for every attempt and randomly selecting the first attempt and then deterministically choosing the remaining selections. The deterministic approaches considered generating the critical pairs in different orders (in particular, generating them in a linearization of the partial order, selecting processes in different orders, and generating the pairs ordered by the first event in the pair rather than the second). Thus far these variations have made little difference to the results that are generated. This suggests at least some degree of robustness in the algorithm. Our current objective is to attempt to determine the quality of the dimension-bound analytically.

4 Results and observations

We have executed our dimension-bound algorithm over several dozen distributed computations covering over half-a-dozen different parallel, concurrent and distributed environments and a range of 3 to 300 processes. The environment types are the Open Software Foundation Distributed Computing Environment [4], the μC++ shared memory concurrent programming language [1], the Hermes distributed programming language [19], the Parallel Virtual

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
45	5	12	3
90	19	27	2
121	20	61	4
249	40	124	3
291	42	164	3
467	42	183	4
297	44	237	4
499	70	443	5
501	72	496	6
833	110	1490	9
817	112	1378	8
928	114	1738	7
902	115	1402	8
1560	159	3579	10

Table 1. Dimension bounds for OSF DCE

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
15	7	38	2
2687	59	2360	4
3252	66	3044	4
2612	66	2032	3
49791	95	6622	3
3272	96	5906	4
7426	109	10019	6
4028	110	8439	6
7928	112	6969	3
35266	112	6675	4
30048	120	7999	3
9826	178	18464	6

Table 2. Dimension bounds for Java

Machine [6] and the Java programming language [7]. Various of the raw results are shown in Tables 1 through 5.

The quick summary is that the dimension bound that we discovered over this range of computations and environments was always 10 or less. For computations of process count greater than 20 there is a minimum of an order of magnitude difference between the dimension and the number of processes. When the number of processes is greater than 100, it is usually a factor of 15 or greater. To help visualize what these results imply, we created a graph, shown in Figure 2, which plots dimension as a function of the number of processes. The two graphs shown are the same, but with differing scales. The horizontal axis is the number of processes while the vertical axis is the dimension bound. We also plot two additional lines. First we show the "dimension = 10" line, as all results were less than or equal to that value. Second, we show the "dimension = width" line,

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
360	12	156	2
1750	12	853	5

Table 3. Dimension bounds for μC++

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
1888	125	1323	5
1944	127	1429	5
4164	267	4403	7
l4086	297	21401	6

Table 4. Dimension bounds for Hermes

which illustrates the increase in Fidge/Mattern vector-clock size as the number of processes increases.

In addition to testing with distributed computations, we created a series of broadcast patterns of varying sizes and crown patterns. The results from our program for these patterns was dimension bound n for the crown patterns and 3 for the broadcasts, regardless the number of processes involved in the broadcast. The dimension of all of these cases is optimal. However, it should by no means be inferred that the algorithm is in general producing optimal results. All we can infer from the current data is that the dimension bounds achieved for a reasonable number and variety of distributed computations is substantially better than the assumed default value of the number of processes involved.

5 Further work

There are several areas in which we are actively working. We have developed an online algorithm for dimension-bound analysis, which we hope will lead us to an online variant of the Ore timestamp. This will not render this offline algorithm obsolete. The offline algorithm is far more useful for checking robustness assumptions under different orderings of the critical pairs (recall that this is the only aspect of the algorithm that is not optimal in terms of computing the dimension). We are also actively attempting to discover the analytical quality of the bounds produced by this algorithm.

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
138	16	270	3
1338	64	4782	5
2682	128	17759	5

Table 5. Dimension bounds for PVM

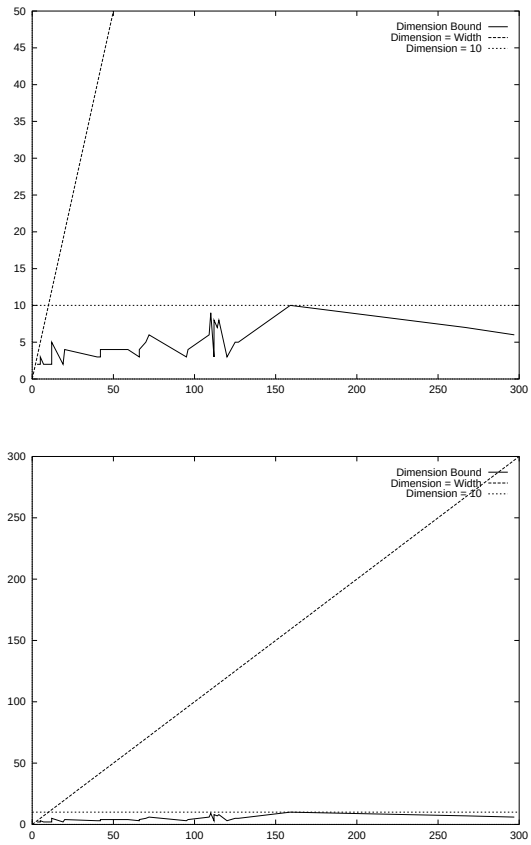


Figure 2. Dimension as a function of number of processes

Acknowledgment

The author would like to thank IBM for supporting this work and David Taylor for many useful discussions.

References

[1] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, Feb. 1992.

[2] B. Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters*, 39:11–16, July 1991.

[3] C. Fidge. Fundamentals of Distributed Systems Observation. Technical Report 93-15, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1993.

[4] O. S. Foundation. *Introduction to OSF DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[5] J. Fowler and W. Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990.

[6] A. Geist, A. Begulin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[8] J. Z. Han. Automatic Comparison of Execution Histories in the Debugging of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.

[9] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[10] C. E. Jaekl. Event-Predicate Detection in the Debugging of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1997.

[11] C. Jard and G.-V. Jourdan. Dependency Tracking and Filtering. Technical Report 851, IRISA, Beaulieu, France, August 1994.

[12] T. Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994.

[13] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. POET: Target-System Independent Visualisations of Complex Distributed-Application Executions. *The Computer Journal*, 40(8):499–512, 1997.

[14] L. Lamport. Time, Clocks and the Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, 1978.

[15] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, December 1988. Elsevier Science Publishers B. V. (North Holland).

[16] O. Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[17] J. L. Sharnowski and B. H. C. Cheng. A Visualization-based Environment for Top-down Debugging of Parallel Programs. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 640–645. IEEE Computer Society Press, 1995.

[18] M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, August 1992.

[19] R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[20] D. J. Taylor. Scrolling Displays of Partially Ordered Execution Histories. In preparation.

[21] W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins University Press, Baltimore, MD, 1992.

[22] M. Yannakakis. The Complexity of the Partial Order Dimension Problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, September 1982.

[23] Y. M. Yong. Replay and Distributed Breakpoints in an OSF DCE Environment. Master's thesis, University of Waterloo, Waterloo, Ontario, 1995.